A Formal Architectural Description Language based on Symbolic Transition Systems and Modal Logic

Pascal Poizat, Jean-Claude Royer

Journal of Universal Computer Science, vol. 12, no. 12 (2006), 1741-1782
submitted: 11/2/05, accepted: 23/12/06, appeared: 28/12/06 © *J.UCS*

A Formal Architectural Description Language based on Symbolic Transition Systems and Modal Logic

Pascal Poizat
(IBISC FRE 2873 CNRS - Université d'Évry Val d'Essonne
Tour Évry 2, 523 place des terrasses, 91000 Évry, France
&
ARLES team, INRIA Rocquencourt
Domaine de Voluceau, B.P. 105, 78153 Le Chesnay Cedex, France
Pascal.Poizat@inria.fr)

Jean-Claude Royer
(OBASCO team, École des Mines de Nantes - INRIA, LINA
4, rue Alfred Kastler, B.P. 20722, 44307 Nantes Cedex 3, France
Jean-Claude.Royer@emn.fr)

Abstract: Component Based Software Engineering has now emerged as a discipline for system development. After years of battle between component platforms, the need for means to abstract away from specific implementation details is now recognized. This paves the way for model driven approaches (such as MDE) but also for the more older Architectural Description Language (ADL) paradigm. In this paper we present KADL, an ADL based on the Korrigan formal language which supports the following features: integration of fully formal behaviours and data types, expressive component composition mechanisms through the use of modal logic, specification readability through graphical notations, and dedicated architectural analysis techniques.

Key Words: Architectural Description Language, Component Based Software Engineering, Mixed Formal Specifications, Symbolic Transition Systems, Abstract Data Types, Modal Logic Glue, Graphical Notations, Verification.
Category: D.2, D.2.1, D 2.2, D 2.10, D.2.11, D.2.13.

1 Introduction

Component Based Software Engineering (CBSE) [Szy98] has now made a breakthrough in software engineering as a discipline for software development which yields promising benefits such as trusted components, assisted component composition and adaptation, increase of the reusability level for software parts and off-the-shelf commercials (COTS). Component middlewares such as CCM [OMG06], .NET [Pla03], J2EE [Sun03], providing the effective means to put components into practice, have proven to be crucial elements in the acceptation of CBSE in the software engineering community. However, a major drawback of the mainstream approach for CBSE was that it was mainly focused on low-level (programming and infrastructures) features, making it difficult to

reason on the problematic issues hidden behind the CBSE promised - and yet not all achieved - results. This was mainly due to the increasing number of component middlewares or frameworks, either general ones (CCM, .NET, J2EE) or specific/extensions ones (*e.g.*, Real-Time CORBA, Lightweight CCM, Fractal). The search to solutions to this *middleware jungle* led to different, yet complementary, proposals such as separation of concerns (Aspect Oriented Programming, Aspect-Oriented Software Design) [KLM+97, FECA05] or Model Driven Engineering (MDE) [Béz05]. They promote the return to the development of abstract models before programs (or implementation specific models) and a clear separation between the functional (business, platform independent) and the more technical or implementation related aspects of software systems.

This need for abstraction, at least in the first steps of the development process, can be adequately supported by Architectural Description Languages (ADL) [MT00], modelling languages focusing on the composition and interaction aspects of component based systems. To design component systems one needs first a *structuring approach* that supports both *decomposition oriented* modelling (decomposing requirements and systems into subparts) and *composition oriented* modelling (building composites from more simple building blocks). ADLs address this issue by providing adequate concepts for the modelling of system architectures, namely components, connectors and configurations.

Components abstract basic composition units of data or computation, with well-specified *interfaces*, made up of interacting points called *ports*, and with explicit dependencies. It is currently accepted that component interfaces can be described at four different levels: signature (provided/required operations or services, with arguments and return values), behaviour (protocols constraining the order of service calls), semantics (of the operations or services) and quality of service (QoS) [CMP06]. *Connectors* are architectural elements modelling interaction (*e.g.*, communication) between components which then play different *roles* for these connectors. *Configurations*, or architectures, are made up of components, connectors and bindings between interfaces (or ports) and roles. Higher-level concepts, such as architectural patterns (reusable abstract configurations) and architectural families or styles (restricting configurations to fixed types of components, connectors and bindings) then build on this simple ADL ontology, described in Figure 1.

Formal methods are mandatory to verify properties of models before transforming them into code. On a wider scale, formal methods provide abstract and non ambiguous model description languages, they are also essential to build tools, to prototype or to test systems. Model driven approaches should integrate some degree of formalism, if not be completely formal. An important feature of ADLs is their formal ground. Without it, ADLs are only *box-and-line* notations. Formal ADLs have proven very efficient to support both the design, verification

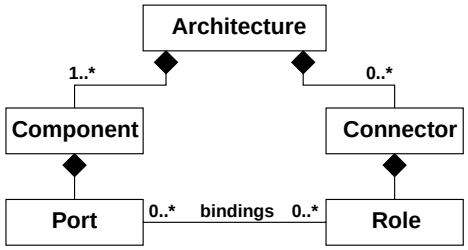


Figure 1: Simplified ADL Metamodel (UML notation)

and deployment activities of software architecture and component based engineering [MT00, BI03, PRS04, CPS06], yet mainly taking into account only the first two component interface levels (signature and behaviours).

With the increase of systems' complexity, the need for a separation of concerns with reference to static (data types, components signature and semantics interfaces) and dynamic (behaviours, communication, components behavioural interfaces) aspects appeared. This issue has been addressed in the formal methods community by the *mixed specification* approach where different aspect models are defined in different specification units. These approaches can be classified into homogeneous and heterogeneous ones [Poi00]. *Homogeneous* approaches encompass all aspects in a single formal framework, *e.g.*, LTL [RL97], rewriting logic [Mes92] or TLA [Lam94]). This makes model integration, definition of consistency criteria and verification easier, but at the cost of a lower expressiveness, adequacy or readability for at least one of the aspects. This is the case with the formal ADLs, *e.g.*, those based on process algebras such as Wright [AG97] or Darwin [MK96]. On the opposite, *heterogeneous* approaches rely on the use of different domain specific languages dedicated to each of the aspects. Examples are LOTOS [ISO89], SDL [IT02] or approaches combining process algebras with the Z specification language (*e.g.*, OZ-CSP [Smi97] or CSP-OZ [Fis97]). This is also reflected in semi-formal modelling languages such as UML [OMG05] where static and dynamic aspects are dealt with by different diagrams (class diagrams, state diagrams, component and interaction diagrams). In these later approaches, the (formal) links and consistency between the aspects are not defined, trivial, or lead to combinatorial problems when verifying the models resulting from the integration of the domain specific ones (the well-known state explosion problem). This limits either the possibilities of reasoning on the whole system or the expressiveness of the formalism. Moreover, whatever is the approach used to deal with mixed specification models, the development of CBSE requires formal specification languages taking also into consideration expressive means to define connections between components in software architectures.

Korrigan [Poi00] is a formal mixed specification language which integrates two domain specific languages: algebraic specifications for the static aspect and Labelled Transition Systems (LTS) for dynamic aspects. Both are integrated into a unifying semantic framework, Symbolic Transition Systems (STS). Communication and interaction between component models in Korrigan is achieved at a high expressiveness level thanks to modal logic.

In [CPR01b] we have presented the principles of the Korrigan model without entering into the detail of its semantics, the focus was rather on a dedicated tool-equipped framework. Verification was there addressed through model transformation, using LOTOS as a possible target language, and thereafter taking advantage of this language tool boxes, *e.g.*, CADP [GLM01]. Prototyping code generation from Korrigan models to Java had been also defined, translating separately static models into pure Java and dynamic models and communication into the Active Java dialect.

Since then, works have addressed extensions of Korrigan, mainly in the CBSE context, and accordingly of its operational and denotational semantics [ABP04, MPR04, PNPR05]. Specification integration principles originating from Korrigan have also made the definition of a generic framework for the integration of formal data types into UML state diagrams possible [APS07]. Our main objective here is to present the *state of the art* status for Korrigan operational semantics and address its relation to CBSE through the definition of KADL, an ADL based on Korrigan principles. KADL inherits STS as the means to describe component behavioural interfaces and modal logic as the way to express component communication.

In [CPR01b], we raised as a perspective the need for verification techniques dedicated specifically to STS. The available verification means for Korrigan, presented in [CPR01b], relied on non symbolic specification languages and their LTS based models, which led to state explosion when using these languages tools on models where data types are not bounded, *e.g.*, integers restricted to $[1, 2, 3]$. This was clearly not satisfactory when verifying interacting components as they may typically present incompatibilities only for some exchanged values (*e.g.*, here 4). KADL is therefore equipped with dedicated verification techniques for STS in the context of component architectural descriptions.

To our knowledge, this is the first approach to propose a formal ADL, dealing with all three first component interface levels (signature, behaviours and semantics), taking into account both static and dynamic aspects, and with dedicated verification techniques avoiding the state explosion problem in presence of non bounded data types. The article is organised as follows. Section 2 first sets formal foundations up and then the KADL formal architectural language is presented in Section 3. Verification techniques are addressed in Section 4. Section 5 details relations between our ADL and existing ones, and finally Section 6

concludes the article. All through the article we use the ATM benchmark case study [DOP00] for explanation and demonstration purposes. The comprehensive case study development in KADL, together with the verification results, can be found in a Technical Report [PR06].

2 Formal Model

In this Section we present the Korrigan language and its operational semantics. They support the definition of the KADL ADL, Section 3 and dedicated verification techniques, Section 4.

2.1 Metamodel

The core of the Korrigan metamodel is presented in Figure 2, using as usual the UML class diagram notation [OMG05]. Concepts (or types) are denoted by classes (boxes). Abstract classes (*e.g.*, *Integration View*) are distinguished from concrete ones (*e.g.*, STS) using *italic*. Relations between concepts are depicted using UML associations (links). Here, two different ones are used: generalization (arrows with a white triangle head) and composition (links with black lozenges). Generalization corresponds to an *is-a* relationship. It is related to the inheritance and subtyping concepts of programming languages. Composition corresponds to a *part-of* relationship. It denotes that instances of a (composite) class are made up of other class instances.

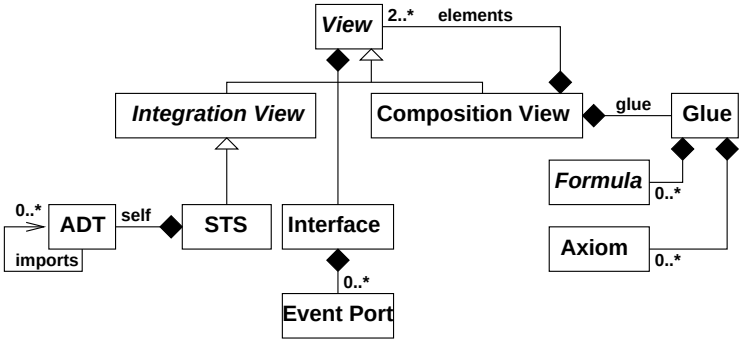


Figure 2: Korrigan Metamodel (UML notation)

Our model is based on the abstract *View* concept which is used to describe in a unifying way the different aspects of a component using integration and composition structuring. A view may either be an *Integration View* or a Composition View. *Views* have a well-defined Interface built on Event Ports. *Integration*

View is an abstraction that expresses the fact that, in order to design a (simple, basic) primitive component, it is useful to be able to express its different aspects (here the static and dynamic aspects, with no exclusion of further aspects that may be identified later on). This integration is concretely achieved using Symbolic Transition Systems (STS) which are transition systems equipped with an underlying Abstract Data Type (ADT), denoted in the transition system using self. *Integration Views* follow a *dynamic encapsulation* principle: their static aspects may only be accessed through their dynamic aspect. Component composition is achieved through Composition Views, made up of several subcomponents (denoted by elements). The subcomponents of a composite can be of any kind of *View*. The coordination or communication between the subcomponents is expressed in a Glue using both algebraic specification Axioms and modal logic *Formulas*.

2.2 Interfaces

Components, both primitive and composite ones, have well-defined interfaces which describe the way they can interact within systems. These interfaces are sets of *event ports*, some form of dynamic signature made up of names with offer parameters (as in LOTOS [ISO89]) and interaction typing (as in SDL [IT02]). A value emission of a D data type is noted !D and a value receipt ?D.

Communication can be synchronous or asynchronous. Moreover, the communication in composites can be hidden or not. Hidden communication is used to ensure that the external environment of composites will not be able to communicate with them using specific (internal, hidden) ports. An event port may correspond either to a provided service (events received from the component environment), to a required service (events sent to the component environment) or to a synchronizing mechanism (rendez-vous, inherited from LOTOS). Only value receipts can be made on provided services, and only value emissions on required ones. Rendez-vous enables one to use both value emissions and value receipts, but it is restricted to synchronous communication.

Interaction typing means that component types may be associated to event ports thanks to the TO and FROM keywords. This enables one to state that any component to be glued on this signature has to satisfy (at least) a given protocol. This makes the support for a simple form of inheritance possible, see 2.3.5

Example 1 System Requirements. In this article we will demonstrate the use of KADL on a formal methods benchmark, the ATM or Till System [DOP00]. This system's requirements are as follows.

The system *is composed of several* tills *which can access a* central resource *containing the detailed records of customers' bank accounts. A* till *is used by inserting a card and typing in a Personal Identification Number (PIN) which is encoded by the till and*

compared with a code stored on the card. After successfully identifying themselves to the system, customers may either (i), make a cash withdrawal or (ii), ask for a balance of their account to be printed. Information on accounts is held in a central database *and may be unavailable in case of network failure. In such a case, actions (i) and (ii) may not be undertaken. If the database is available, any amount up to the total in the account may be withdrawn. Withdrawals are subject to a daily limit, which means that the total amount withdrawn within a day has to be stored on cards. Daily limits are specific to each customer and are part of their bank account records. Another restriction is that a withdrawal amount may not be greater than the value of the till local stock.*

Tills may keep "illegal" cards, i.e., cards which have failed a key checking. Each till is connected to the central *by a specific* line*, which may be up or down. The central handles multiple and concurrent requests. Once a user has initiated a transaction, it is eventually completed and preferably within some real time constraint. A given account may have several cards authorized to use it.*

With reference to the original case study we add the following extensions: a management of the local stock of cash within tills, and a daily limit which is specific to each customer. In 3.5, we will also consider an extension of the network using a multiplexer. We do not take into account the real time constraints.

Example 2 Interface. The *interface* of a till is the following one. Inputs (provided services) are card ?Card to insert the user card, pin ?PinNumber to enter the PIN, getSum ?Money to enter the desired cash amount, add ?Money to allow an operator to add money to the till available amount, and rec ?Msg to receive a message from the bank connection. Outputs (required services) are card !Card to eject the card, cash !Money to give money, and send !Msg to send a message on the bank connection.

2.3 Integration Views

Primitive components are sequential components described with two aspects: a data type description of their functional operations and a behavioural protocol. Both are integrated within an integration view. In this section we will present a concrete instantiation of integration views, namely Symbolic Transition Systems (STS).

2.3.1 Data Type Models

Data types models are defined using algebraic specifications, yielding Algebraic Data Types (ADT). Model oriented specifications (B machines or Z schemas) may be used in replacement or in conjunction with these algebraic specifications following the [APS07] principles. In [PNPR05] we have also done experiments using Java classes.

Here we give only the necessary insight into algebraic specifications. More details are given in [AKBK99]. Algebraic specifications abstract concrete implementation languages such as Java, C++, or Python. A *signature* (or static interface) Σ is a pair $(\mathcal{S}, F)$ where $\mathcal{S}$ is a set of *sorts* (type names) and F a set of function names equipped with *profiles* over these sorts. If R is a sort, then Σ_R denotes the signature $(\mathcal{S}, F_R)$, with F_R the subset of functions from F with result sort being R. X is used to denote the set of all variables. From a signature Σ and from X, one may obtain *terms*, denoted by $T_{\Sigma,X}$. The set of *closed terms* (also called ground terms) is the subset of $T_{\Sigma,X}$ without variables, denoted by T_Σ. An *algebraic specification* is a pair (Σ, Ax) where Ax is a set of axioms between terms of $T_{\Sigma,X}$. $r\downarrow$ denotes the normal form (assumed to be unique) of the ground term r. $[R]$ denotes the set of all normal form terms of sort R and $r : R$ means that r has type R. $r(u)$ denotes the application of r to u.

Example 3 Data Type Model (ADT). The Card ADT, Figure 3, is used to specify the properties of cards which are read by the tills.

```
Sort Card
Imports Boolean, Natural, PinNumber, Money, Card, Ack, Info

Opns
/* generator of Card */
newCard          : Ident x Money x Money x PinNumber x Date -> Card
/* other Card constructors */
noCard           : Card                          /* no card */
updateDailyLimit : Card x Money x Date -> Card /* update daily limit */
/* accessors */
id               : Card                -> Ident     /* client id */
max              : Card                -> Money     /* daily limit */
sum              : Card                -> Money     /* daily amount */
code             : Card                -> PinNumber /* PIN code */
last             : Card                -> Date      /* last withdraw */

/* axiom variables */
Variables i:Ident; m,s,s1:Money; c:PinNumber; d,d1:Date

Axioms
id  (newCard(i, m, s, c, d)) = i
max (newCard(i, m, s, c, d)) = m
sum (newCard(i, m, s, c, d)) = s
code(newCard(i, m, s, c, d)) = c
last(newCard(i, m, s, c, d)) = d
d=d1  => updateDailyLimit(newCard(i, m, s,    c, d), s1, d1)
      =                   newCard(i, m, s+s1, c, d)
d!=d1 => updateDailyLimit(newCard(i, m, s,    c, d), s1, d1)
      =                   newCard(i, m, s1,   c, d1)
```

Figure 3: Card ADT

In addition to the ADT of data types used in components or exchanged in communications, a *specific ADT* is given for each component. It describes the

semantics of the functional operations of the component. The behavioural model of components (their transitions) may use this ADT to denote relations between the component interactions (inputs or outputs events) and its operations. A dedicated sort corresponding to the STS is called *sort of interest*, and a term of this sort is denoted by self in the STS. We here consider a simple approach where (i) actions are explicitly given in behaviours and then defined in the ADT, and (ii) the axioms are fully given by the specifier. In [Roy03] we present a more automated approach which enables one to derive part of the operations semantics, namely profiles and left-hand part of axioms, from the behaviours.

Example 4 Component ADT. The Till ADT, Figures 4 and 5, is the ADT which defines the functional operations available in the tills. We omit the imported ADT (but for Card, Fig. 3) due to lack of place.

```
Sort Till
Imports Boolean, Natural, PinNumber, Money, Card, Ack, Info

Opns
/* generator of Till */
newTill    : Money x Card x PinNumber x Money x Date x Natural -> Till
/* other Till constructors */
addCash    : Till x Money     -> Till /* cash adding */
insertCard : Till x Card      -> Till /* card insertion */
pin        : Till x PinNumber -> Till /* PIN entry */
getSum     : Till x Money     -> Till /* withdrawal choice */
giveCash   : Till             -> Till /* cash output */
keepCard   : Till             -> Till /* keeping the card */
giveCard   : Till             -> Till /* returning the card */
clock      : Till             -> Till /* clock increasing */

/* accessors */               /* other observers */        /* guards */
amount  : Till -> Money       msgValidity : Till -> Info   pinOK : Till      -> Boolean
card    : Till -> Card                                     retry : Till      -> Boolean
code    : Till -> PinNumber                                fail  : Till      -> Boolean
sum     : Till -> Money                                    check : Till      -> Boolean
date    : Till -> Date                                     ack   : Till, Ack -> Boolean
counter : Till -> Natural
```

Figure 4: Till ADT (operations)

2.3.2 Behavioural Models

The behavioural protocols of components, integrating these components data types, are described in a finite way thanks to Symbolic Transition Systems (STS). STS have appeared under different forms in the literature [IL01, CMS02, CPR00, JJRZ05] as a solution to the state (and transition) explosion problem. However, this problem is essentially not a problem of modelling and specifying systems, but one of verifying such systems, mainly with model checking. In this Section,

```
Variables
a,sum,a2,sum2:Money; c,c2:Card; code,code2:PinNumber; r:Ack; today,today2:Date ; cpt:Natural;
self:Till

Axioms

addCash   (newTill(a,c,code,sum,today,cpt),a2)   = newTill(a+a2,c,code,sum,today,cpt)
insertCard(newTill(a,c,code,sum,today,cpt),c2)   = newTill(a,c2,code,sum,today,0)
pin       (newTill(a,c,code,sum,today,cpt),code2)= newTill(a,c,code2,sum,today,cpt+1)
getSum    (newTill(a,c,code,sum,today,cpt),sum2) = newTill(a,c,code,sum2,today,cpt)
giveCash  (newTill(a,c,code,sum,today,cpt))      =
                  newTill(a-sum,updateDailyLimit(card(self),sum,today),code,sum,today,cpt)
giveCard  (newTill(a,c,code,sum,today,cpt))      = newTill(a,noCard,code,sum,today,cpt)
keepCard  (newTill(a,c,code,sum,today,cpt))      = newTill(a,noCard,code,sum,today,cpt)
clock     (newTill(a,c,code,sum,today,cpt))      = newTill(a,noCard,code,sum,inc(today),cpt)

/* constant for initialisation */
new = newTill(0,noCard,0,0,0,0)

/* accessors */
amount  (newTill(a,c,code,sum,today,cpt)) = a
card    (newTill(a,c,code,sum,today,cpt)) = c
code    (newTill(a,c,code,sum,today,cpt)) = code
sum     (newTill(a,c,code,sum,today,cpt)) = sum
date    (newTill(a,c,code,sum,today,cpt)) = today
counter (newTill(a,c,code,sum,today,cpt)) = cpt

/* compute the message to allow withdraw */
msgValidity(self) = newInfo(id(card(self)),sum(self))

/* pin code control: ok, wrong and wrong+3 tests */
pinOK(self) =     equals(crypt(code(self)), code(card(self))) AND counter(self) <= 3
retry(self) = NOT equals(crypt(code(self)), code(card(self))) AND counter(self) <  3
fail(self)  = NOT equals(crypt(code(self)), code(card(self))) AND counter(self) >= 3

/* local controls of the till and the card */
check(self) = (sum(self) <= amount(self))
              AND (
                    (equals(last(card(self)), date(self))
                  AND
                    (sum(card(self)) + sum(self) <= max(card(self))))
                OR
                    (NOT equals(last(card(self)), date(self))
                  AND
                    (sum(self) <= max(card(self))))
              )

/* acknowledgment from the bank */
ack(self,r) = isOk(r)
```

Figure 5: Till ADT (axioms)

we focus on modelling issues. Verification will be addressed in Section 4. The description of an STS may be given either in a graphical form (Fig. 6) or in a textual form [PNPR05] (better suited for tool processing).

Example 5 Behavioural Model (STS). The behavioural model of tills is given in Figure 6.

A transition such as cash !sum / giveCash(self) means that the till emits a sum of money. After this, the giveCash operation is used to decrease correspondingly the amount of money available in the till and update the daily limit of the card

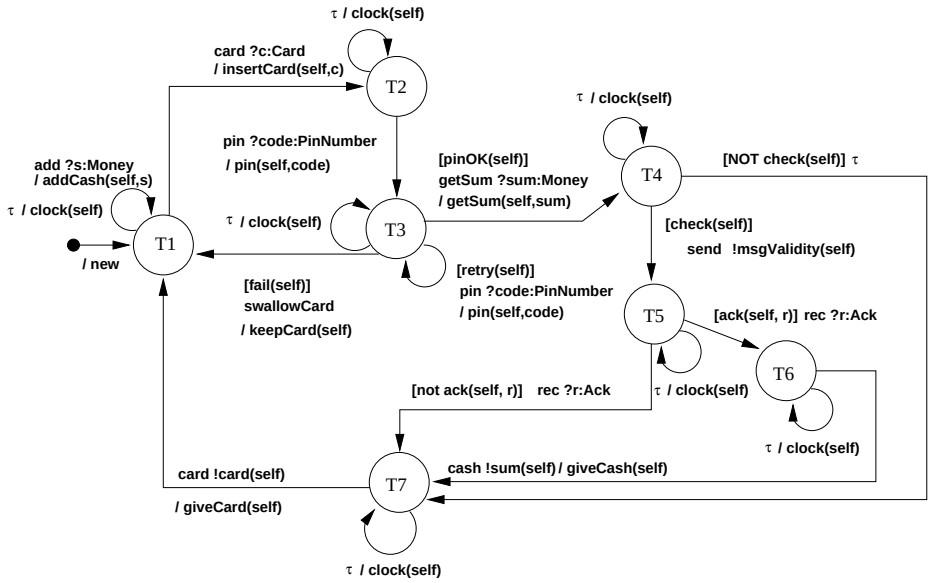


Figure 6: Till Behavioural Model (STS)

as specified by the giveCash axiom, Figure 5.

Our STS have the following features. First of all, they rely on both a static description of a data type (denoted by self) and a dynamic event-oriented one. Transitions have the form: [guard] event / action. guard is a predicate on self and possibly received values which has to yield true for the transition to be fireable. event is a communication event (a communication port name together with reception variables denoted using ? or/and emission terms denoted using !). action is the action to be done when the transition is fired. Both the guard and the action part can be empty. A τ action can be used to denote non observable (internal) events. In Figure 6 this is used for example to denote the passing of time.

The main interest with these transition systems is that (i) using open terms in transitions (received variables), they ensure conciseness (and finiteness) of behavioural models, and (ii) using an open term in states (self), they define equivalence classes (one per state) and hence strongly relate the dynamic and the static (algebraic) representation of a data type.

Definition 1 STS. An STS is a tuple $M = (D, (\Sigma, Ax), S, L, s^0, T)$ where:

- (Σ, Ax) is an algebraic specification,
- D is a sort called *sort of interest* defined in (Σ, Ax),

- $S = \{s_i\}$ is a countable set of states,
- $L = \{l_i\}$ is a countable set of event labels,
- $s^0 \in S$ is the initial state, and
- $T \subseteq S \times T_{\Sigma_{Boolean},X} \times Event(L) \times T_{\Sigma_D,X} \times S$ is a set of transitions.

Note that *countable* means that the set may be infinite but can be enumerated.

Events ($Event(L)$) denote atomic activities that occur in the components. Events are either: *i*) hidden (or internal) events: τ, *ii*) silent events: l, with $l \in L$, *iii*) emissions: $l!e$, with $e \in T_{\Sigma,\{\mathsf{self}\}}$ and $l \in L$, or *iv*) receptions: $l?x : R$ with $x \in X \backslash \{\mathsf{self}\}$, R a sort available from (Σ, Ax) and $l \in L$. Internal events denote internal actions of the components which may have an effect on its behaviour, yet without being observable from its context. Silent events are pure synchronizing events, while emissions and receptions naturally correspond, respectively, to requested and provided services of the components. To simplify we only consider binary communications here, but emissions and receptions may be extended to n-ary emissions and receptions. STS transitions are tuples $(s, \mu, \epsilon, \delta, t)$ for which s is called the source state, t the target state, μ the guard, ϵ the event and δ the action. Each action is denoted by a term with variables where at least self occurs. A do-nothing action is simply denoted by self. In the forthcoming figures, transitions will be labelled as follows: $[\mu]\ \epsilon\ /\ \delta$.

2.3.3 Semantics

The semantics of STS is formalised using configuration graphs. They are obtained applying both the unfolding of receptions and the reduction of ground terms to their normal forms. The process of reducing a ground term is an abstract and operational way to denote term evaluation, the value being the normal form.

Definition 2 Unfolding. The unfolding of an STS $M = (D, (\Sigma, Ax), S, L, s^0, T)$, in $v^0 \in T_{\Sigma_D}$, is the STS $G(M, v^0) = (D, (\Sigma, Ax), S', L, (s^0, v^0\downarrow), T')$. The sets $S' \subseteq S \times D$ and T' are inductively defined by the rules:

- $(s^0, v^0\downarrow) \in S'$
- for each $(s, v) \in S'$
  - if $(s, \mu, \tau, \delta, t) \in T$ and $\mu(v)\downarrow = true$ then $s' = (t, \delta(v)\downarrow) \in S'$ and $((s, v), true, \tau, \mathsf{self}, s') \in T'$;
  - if $(s, \mu, l, \delta, t) \in T$ and $\mu(v)\downarrow = true$ then $s' = (t, \delta(v)\downarrow) \in S'$ and $((s, v), true, l, \mathsf{self}, s') \in T'$;

- if $(s, \mu, l!e, \delta, t) \in T$ and $\mu(v)\downarrow = true$ then $s' = (t, \delta(v)\downarrow) \in S'$ and $((s, v), true, l!e(v)\downarrow, \mathsf{self}, s') \in T'$;
- if $(s, \mu, l?x : R, \delta, t) \in T$ then for each $r : R$ such that $\mu(v, r)\downarrow = true$, there is $s' = (t, \delta(v, r)\downarrow)) \in S'$ and $((s, v), true, l!r, \mathsf{self}, s') \in T'$.

Pairs (s, v) are called *configurations* where s is the *control state*. Let M be an STS. Its unfolding in a v^0 term, $G(M, v^0)$, is called a *configuration graph*. A configuration graph is a particular STS without reception, where guards are all equal to *true*, emission terms are in normal form and actions are do-nothing actions denoted by self.

2.3.4 Comparison with Statecharts

STS may be (graphically speaking) related to Statecharts [Har87] (or UML state diagrams [OMG05]). However, there are differences:

– *Syntactically*, Statecharts are more expressive as STS have no concurrent states, histories, stub or junctions. Histories can be achieved using specific variables in the data type underlying the STS. STS have no need for concurrent states: an STS is a sequential component; concurrency, and communication, are achieved through composition using several communicating STS and the computation of a structured STS from subcomponent STS, see Section 2.4. Several hierarchical state notations have been defined for our STS [Poi00] but they are still more simple than the Statecharts ones (*e.g.*, transition crossing over states is forbidden) to avoid exponential complexity when flattening the behavioural models which is a drawback of Statecharts when verification is the issue [DMY02].

– *Semantically*, communication between STS is basically synchronous. However, we have been able to take into account asynchronous communication (which is important in a component based context) in STS either using specific variables in the underlying data type [MPR04] or directly in the STS semantics, which then gets more complicated [APS07]. Therefore, we choose here to present the purely synchronous communication semantics to be more simple. Concurrency is asynchronous in STS (as in process algebras such as CSP or LOTOS). In contrast to this, Statecharts rely on a more complex *run to completion step* principle. There is a strong link between an STS and its corresponding data type. An STS may be seen as a graphical representation of an abstract interpretation of an algebraic data type [Roy03]. More generally the integration of STS with formal data types is formally well defined.

– From the *design process point of view*, STS may be built using conditions which enable one to semi-automatically derive them from requirements, as presented in [CPR99].

2.3.5 Inheritance

In software engineering, inheritance is one of the key concepts that enable reuse. Inheritance enables one to add operations (or methods), and to add or strengthen constraints. It may also be used to perform overloading and masking. Here, a simple form of inheritance is provided for STS. Inheritance mechanisms are restricted to the adding of new states and new transitions in behaviours. Overloading and masking are forbidden since they yield semantic complexity. These rather strict inheritance constraints however simplify the dynamic descriptions of views, allow subtyping and ensure some kind of behavioural compatibility. Here, inheritance semantically corresponds to trace inclusion (the traces of the super-view are a subset of the traces of the sub-view). More complex behavioural relations could be used, *e.g.*, [Nie95, Sou96, PV02, CVZ06] but would have to be extended for STS.

2.4 Composition Views

Integration views are used to define atomic primitive components. Composition views are introduced to define composites as a set of (sub)components (either primitive or composite ones) and to explicit composition constraints between these subcomponents.

2.4.1 Composition Models

Compositions (or composites) are made up of a set of identified (over some domain Id) subcomponents, $\{i : M_i, i \in \mathrm{Id}\}$, and glue rules which enable one to specify relations between the subcomponents. This glue is made up of:

- a set of axioms, Ax_Θ, to define relations between the subcomponents functional operations, between one subcomponent guard and another subcomponent operation or relations between the composite functional operations and its subcomponents ones. More generally, these axioms are used to denote constraints on the composite static aspect;
- modal logic state formulas to define relations between the subcomponents behaviours or relations between the composite behaviour and its subcomponents ones. These formulas are:
  - a modal logic state formula, $\overline{\psi}$[1], to impose constraints (*e.g.*, an invariant) on the states of the composite, possibly in terms of the states of the subcomponents;

[1] In the sequel, we use over-lined notations, *e.g.*, $\overline{S}$, to denote elements related to a composition.

- a modal logic state formula, $\overline{\psi_{\mathrm{I}}}$, to impose additional constraints on initial states;
- a set of modal logic transition formulas, $\overline{\lambda}$, to impose communication related constraints in terms of the transitions of the subcomponents.

More generally, these formulas are used to denote constraints on the composites dynamic aspect. The $\overline{\lambda}$ set corresponds typically to the bindings in usual ADLs, yet being more expressive (see 2.4.3 below), while state formulas and axioms have no counterpart in such ADLs.

Modal Logics. Modal logics are very powerful means to express properties of systems. Among them, different temporal logics exist. These logics deal with evolving truth values (using temporal modal operators) for properties on states of the dynamic models. There is a first separation between linear (LTL) and branching time (CTL, CTL*) logics. Such temporal logics have then been extended to take actions into account (ACTL, TLA, HML). We advocate that these logics are also expressive means to coordinate entities. Note that to keep the computation of the semantics of composites simple, we restrict like HML to a logic were the temporal modalities are limited to the next operators (EX and AX in CTL). State formulas correspond to invariants (AG in CTL). The idea is to be able first to denote whole sets of objects that are to be glued (in the STS models these objects are states and transitions). This denotation is achieved using logic formulas, with a set-theoretic semantics: given a set S and a formula f, the semantics of f over S is the subset of S elements which satisfy f. Then, the logic must also take into account coordination and be able to lift the properties of the subcomponents of a composition up to the composition. This is achieved using indexed formulas.

Syntax. We first define means to express the properties of transitions. This is achieved using *transition formulas* which are defined as:

$$\lambda ::= \mathbf{true} \mid \mathrm{l} \mid \neg\lambda \mid \lambda_1 \wedge \lambda_2$$

where a l is an event pattern, *i.e.*, an event name and direction (*e.g.*, $e?$ or $e!$). Such patterns are used to match a subset of the transitions. In our model, patterns correspond to transition labels (*e.g.*, pattern $e?$ matches all the $e?$ transitions). Properties of states are expressed using *state formulas*:

$$\psi ::= P \mid \mathbf{true} \mid @s \mid [\lambda]\psi \mid \neg\psi \mid \psi_1 \wedge \psi_2$$

where P is a state property (any first order formula which is defined over self in the $\Sigma_{\mathrm{Boolean}}$ of the STS the formula will be applied to), s a state identifier, and λ a transition formula. Temporal modalities are inspired from HML. $[\lambda]\psi$ means that any outsourcing transition that satisfies λ will lead to a state that satisfies

ψ. $\langle\lambda\rangle\psi$, which may be defined as $\neg[\lambda]\neg\psi$, means that there is one outsourcing transition that satisfies λ which leads to a state that satisfies ψ.

Transition and state formulas express properties of basic entities. They are then lifted to compositions (and hence glue) using *composition-oriented transition formulas* and *composition-oriented state formulas*:

$$\overline{\lambda} ::= \mathbf{true} \mid c.\lambda \mid \neg\overline{\lambda} \mid \overline{\lambda}_1 \wedge \overline{\lambda}_2 \qquad \overline{\psi} ::= \mathbf{true} \mid c.\psi \mid \neg\overline{\psi} \mid \overline{\psi}_1 \wedge \overline{\psi}_2$$

with c being the identifier of one of the composition components. In any type of formula, $\vee$, $\Rightarrow$, and $\Leftrightarrow$ can be defined as usual.

Compositions may use a syntactic sugar: the *range* operator. This is a way to denote a set of values, either as an interval (i:[1..N]) or by enumerating the values (i:{e_1,...,e_n}). The range operator may be associated either with a universal quantifier ($\forall$) or with a disjunction quantifier ($\oplus$). With ϕ being any kind of formula and $\phi[v/i]$ denoting the substitution of i by v in ϕ, their meaning is the following one:

$$\forall i : [1..N]\phi_i \Leftrightarrow \phi[1/i] \wedge \ldots \wedge \phi[N/i]$$

$$\begin{aligned}\oplus i : [1..N]\phi_i \Leftrightarrow\ & (\phi[1/i] \wedge \neg\phi[2/i] \wedge \ldots \wedge \neg\phi[N/i]) \\ & \vee \ldots \vee \\ & (\neg\phi[1/i] \wedge \neg\phi[2/i] \wedge \ldots \wedge \phi[N/i])\end{aligned}$$

For example, the following formula:

$$\forall i : [1..N](server.send \Leftrightarrow client.i.receive)$$

is a shorthand for:

$$(server.send \Leftrightarrow client.1.receive) \wedge \ldots \wedge (server.send \Leftrightarrow client.N.receive)$$

Examples of our modal logic formulas, used to glue components, are given in 2.4.3.

Definition 3 Composite. A composite is a tuple $C = (\mathrm{Id}, \{i : M_i, i \in \mathrm{Id}\}, Ax_\Theta, \overline{\psi}, \overline{\psi_\mathrm{I}}, \overline{\lambda})$ where:

- Id is a set of identifiers,
- $\{i : M_i, i \in \mathrm{Id}\}$ is a set of STS (Def. 1),
- Ax_Θ is a set of axioms,
- $\overline{\psi}$ and $\overline{\psi_\mathrm{I}}$ are composition state formulas, and
- $\overline{\lambda}$ is a set of composition transition formulas.

2.4.2 Semantics

The semantics of transition and state formulas are defined given a model (an STS) $M =< D, (\Sigma, Ax), S, S_0, v_0, T >$ and denote respectively sets of transitions and sets of states. In the following, we use $S(M)$ to denote the S part of an STS model M, $T(M)$ to denote its T part, label(t) to denote the label of a t transition, source(t) to denote its source state, and target(t) to denote its target state.

The semantics of transition formulas is defined as:

$$\|\mathbf{true}\|_M = T(M)$$
$$\|\mathrm{l}\|_M = \{t \in T(M) \mid \mathrm{label}(t) = \mathrm{l}\}$$
$$\|\neg\lambda\|_M = \|\mathbf{true}\|_M \backslash \|\lambda\|_M$$
$$\|\lambda_1 \wedge \lambda_2\|_M = \|\lambda_1\|_M \cap \|\lambda_2\|_M$$

true denotes all transitions, whereas a formula l denotes only the transitions that have this label. Then, other formulas denotations are obtained from these two base definitions using a set-theoretical approach (difference for $\neg$ and intersection for $\wedge$). Given a model M, a transition t of this model and a transition formula λ, we have $M \models_t \lambda$ iff $t \in \|\lambda\|_M$ and $M \models \lambda$ iff $M \models_t \lambda$ for every t in $T(M)$.

The semantics of state formulas is defined in the same way as:

$$\|P\|_M = \{s \in S \mid P(s)\}$$
$$\|\mathbf{true}\|_M = S(M)$$
$$\|@s\|_M = s$$
$$\|[\lambda]\psi\|_M = \{s \in S(M) \mid \forall t \in \|\lambda\|_M \,.\, \mathrm{source}(t) = s \Rightarrow \mathrm{target}(t) \in \|\psi\|_M\}$$
$$\|\neg\psi\|_M = \|\mathbf{true}\|_M \backslash \|\psi\|_M$$
$$\|\psi_1 \wedge \psi_2\|_M = \|\psi_1\|_M \cap \|\psi_2\|_M$$

A state property denotes all states for which it is true. A formula $[\lambda]\psi$ denotes (source) states such that all (target) states related by transitions of the λ denotation are in the ψ denotation. Other denotations are defined in the same way than transition formulas. Given a model M, a state s of this model and a state formula ψ, $M \models_s \psi$ iff $s \in \|\psi\|_M$ and $M \models \psi$ iff $M \models_s \lambda$ for every s in $S(M)$.

A binding formula on state, $\overset{@}{\exists}\, x.\psi$ with x being a variable and ψ a state formula, is also defined. Its semantics given a model M and a state s is $M \models_s \overset{@}{\exists}\, x.\psi$ iff $M \models_s \psi[s/x]$, where $\psi[s/x]$ denotes the substitution of x by s in ψ. The following formula, $\overset{@}{\exists}\, x.\langle a\rangle @x$, denotes for example all states for which there is a loop transition labelled by a.

We may now define global models obtained from a composition. In a first step, we do not take into account the gluing properties, that is we define a free global model.

Definition 4 Free Global Model. Given a composite $C = (\mathrm{Id}, \{j : M_j, j \in \mathrm{Id}\}, Ax_\Theta, \overline{\psi}, \overline{\psi_\mathrm{I}}, \overline{\lambda})$, with $M_{j\in\mathrm{Id}} =< D_j, (\Sigma_j, Ax_j), S_j, L_j, {s^0}_j, T_j >$, a *free global model* for C is an STS $\overline{M}_{\mathrm{free}}(C) =< \overline{D}, (\overline{\Sigma}, \overline{Ax}), \overline{S}, \overline{L}, \overline{s^0}, \overline{T} >$ such that:

- $\overline{D}$ and $(\overline{\Sigma}, \overline{Ax})$ are obtained from a product type of the ADT specification, adding the Ax_Θ glue axioms within;
- $\overline{S} = \Pi_{j\in\mathrm{Id}}\ j.S_j$, $\overline{L} = \Pi_{j\in\mathrm{Id}}\ L_j$, $\overline{s^0} = \Pi_{j\in\mathrm{Id}}\ j.{s^0}_j$;
- $\overline{T} \subseteq \overline{S} \times T_{\Sigma_{\mathrm{Boolean}},X} \times \Pi_{j\in\mathrm{Id}}(\mathrm{Event}(L_j) \cup \{\varepsilon\}) \times T_{\Sigma_{\overline{\mathrm{D}}},X} \times \overline{S}$ is defined as follows:
  $(\Pi_{j\in\mathrm{Id}} j.s_j, \bigwedge_{j\in\mathrm{Id}} \mu_j, \Pi_{j\in\mathrm{Id}} l_j, \Pi_{j\in\mathrm{Id}} \delta_j, \Pi_{j\in\mathrm{Id}} j.t_j) \in \overline{T}$ iff for every j in Id
  - if $l_j = \varepsilon$ then $s_j = t_j$, $\mu_j = \mathbf{true}$ and $\delta_j = \mathsf{self}$;
  - otherwise there is a transition $(s_j, \mu_j, l_j, \delta_j, t_j)$ in T_j.

ε is used to denote a do-nothing event, *i.e.*, a subcomponent which does not participate in a synchronizing.

A free global composition model is made up of indexed state and transition products built from states and transitions of the composition subcomponents. Indexing is used to be able to denote, at the composition level, a property of a given (named) subcomponent. Apart from this, an interesting property of global composition models is that they have the same transition system structure than component models, hence composites can be used as components in larger compositions. In the following, we use $\overline{S}(\overline{M})$ to denote the $\overline{S}$ part of a $\overline{M}$ model, $\overline{s^0}(\overline{M})$ to denote its $\overline{s^0}$ part, $\overline{T}(\overline{M})$ to denote its $\overline{T}$ part, label$(\overline{t})$ to denote the label of a $\overline{t}$ transition, source$(\overline{t})$ to denote its source state, and target$(\overline{t})$ to denote its target state. Moreover, we define the set of identifiers of a composition model as $\overline{\mathrm{Id}}(\overline{M}) = \{j \mid \exists \overline{s} \in \overline{S}(\overline{M})\, \exists j.s \in \overline{s}\}$. Suffixes are used to denote the projection of a (global) element on a given identifier (*e.g.*, given a $\overline{t}$ transition, $\overline{t}_c$ corresponds to the transition of c which has been used to build $\overline{t}$).

Coordination using temporal logic is given using a set $\overline{\lambda}$ of composition-oriented transition formulas (all possibly **true**), and two $\overline{\psi}$ and $\overline{\psi_\mathrm{I}}$ composition-oriented state formulas (both possibly **true**). The semantics of composition-oriented transition formulas is defined on free global models in the following way:

$$\begin{aligned}
&\|\mathbf{true}\|_{\overline{M}} = \overline{T}(\overline{M})\\
&\|c.\lambda\|_{\overline{M}} = \{\overline{t} \in \overline{T}(\overline{M}) \mid \exists c \in \overline{\mathrm{Id}}(\overline{M})\ \overline{t}_c \in \|\lambda\|_{\overline{M}_c}\}\\
&\|\neg\overline{\lambda}\|_{\overline{M}} = \|\mathbf{true}\|_{\overline{M}} \backslash \|\overline{\lambda}\|_{\overline{M}}\\
&\|\overline{\lambda}_1 \wedge \overline{\lambda}_2\|_{\overline{M}} = \|\overline{\lambda}_1\|_{\overline{M}} \cap \|\overline{\lambda}_2\|_{\overline{M}}
\end{aligned}$$

The interesting part here is the indexed formulas $(c.\lambda)$ which are true for a given $\overline{t}$ transition of the global model only if there is a subcomponent identified by c in

the global model and if λ is true for this subcomponent part of the $\overline{t}$ transition ($\overline{t}_c$). Given a model $\overline{M}$, a transition $\overline{t}$ of this model, and a formula $\overline{\lambda}$, $\overline{M} \models_{\overline{t}} \overline{\lambda}$ iff $\overline{t} \in \|\overline{\lambda}\|_{\overline{M}}$, and $\overline{M} \models \overline{\lambda}$ iff $\overline{M} \models_{\overline{t}} \overline{\lambda}$ for every $\overline{t}$ in $\overline{T}(\overline{M})$. Composition-oriented state formulas are handled in the same way, and the $\overset{@}{\exists}$ operator defined earlier on state formulas can be lifted to composition-oriented state formulas: given a model $\overline{M}$ and a state $\overline{s}$ of this model, $\overline{M} \models_{\overline{s}} \overset{@}{\exists}\, \overline{x}.\overline{\psi}$ iff $\overline{M} \models_{\overline{s}} \overline{\psi}[\overline{s}/\overline{x}]$. Other formula operators can be lifted up to composition formulas too, such as the $[\lambda]\psi$ state formula operator:

$$\|[\overline{\lambda}]\overline{\psi}\|_{\overline{M}} = \{\overline{s} \in \overline{S}(\overline{M}) \mid \forall \overline{t} \in \|\overline{\lambda}\|_{\overline{M}} \,.\, \text{source}(\overline{t}) = \overline{s} \Rightarrow \text{target}(\overline{t}) \in \|\overline{\psi}\|_{\overline{M}}\}$$

We may now define our glued global models (or global models for short).

Definition 5 Global Model. Given a composite $C = (\text{Id}, \{j : M_j, j \in \text{Id}\}, Ax_\Theta, \overline{\psi}, \overline{\psi}_\text{I}, \overline{\lambda})$, with $M_{j\in\text{Id}} =< D_j, (\Sigma_j, Ax_j), S_j, L_j, {s^0}_j, T_j >$, a *global model* for C is an STS $\overline{M}(C) =< \overline{D}, (\overline{\Sigma}, \overline{Ax}), \overline{S}, \overline{L}, \overline{s^0}, \overline{T} >$ such that:

- let $\overline{M}_\text{free}(C) =< \overline{D}_\text{free}, (\overline{\Sigma}_\text{free}, \overline{Ax}_\text{free}), \overline{S}_\text{free}, \overline{L}_\text{free}, \overline{s^0}_\text{free}, \overline{T}_\text{free} >$ be the free global model of C,
- $\overline{D} = \overline{D}_\text{free}$ and $(\overline{\Sigma}, \overline{Ax}) = (\overline{\Sigma}_\text{free}, \overline{Ax}_\text{free})$,
- $\overline{S} = \{\overline{s} \in \overline{S}_\text{free} \mid \overline{M}_\text{free} \models_{\overline{s}} \overline{\psi}\}$,
- $\overline{L} = \overline{L}_\text{free}$,
- $\overline{s^0} = \{\overline{s^0} \in \overline{s^0}_\text{free} \mid \overline{M}_\text{free} \models_{\overline{s^0}} \overline{\psi} \wedge \overline{\psi}_\text{I}\}$,
- $\overline{T} = \{\overline{t} \in \overline{T}_\text{free} \mid \forall \overline{\lambda_i} \in \overline{\lambda},\ \overline{M}_\text{free} \models_{\overline{t}} \overline{\lambda_i}\}$.

This semantics is supported by the ETS plug-in [Poi05]. It can be obtained in an *on-the-fly* way to be more efficient. Thus in practice, we reduce for example $\overline{S}$ to be the set of reachable states from $\overline{s^0}$.

2.4.3 Expressiveness of the Glue

Our glue is expressive and enables us to link symbolic transition systems to denote synchronizations and communications in an abstract and concise way. It may also denote complex synchronizations between sequences of states or transitions as in Figure 16. Here are some examples of basic communication patterns which can be expressed:

- synchronizing, *e.g.*, c_1 and c_2 must synchronize on a: $c_1.a \Leftrightarrow c_2.a$;
- event ports connection (ports may have different names): $c_1.a_1 \Leftrightarrow c_2.a_2$;

- broadcasting (1 to N): $\forall i : [1..N](server.send \Leftrightarrow client.i.receive)$;
- exclusive peer to peer (1 to 1): $server.send \Leftrightarrow \oplus i : [1..N](client.i.receive)$;
- as in LOTOS, value passing if glued labels are of sort ?/!, value agreement if glued labels are of sort !/! (*e.g.*, two components synchronizing on a given frequency), and value negotiation if glued labels are of sort ?/? (*e.g.*, a given arbitrary number is chosen by two components before going on communicating); for more details on value agreement and value negotiation, see [ISO89];
- exclusive states, *e.g.* c_1 and c_2 may not be in their *on* state at the same time: $\neg(c_1.@on \wedge c_2.@on)$;
- composite event exportation: $self.a \Leftrightarrow self.sub.a$, where $self$ denotes the current component, being composed of a *sub* subcomponent.

3 The KADL Architectural Description Language

In this Section we present the KADL ADL and demonstrate its use on a specification benchmark [DOP00]. We first present simple primitive components and then architectures. Note that the overall architectural description of the case study is presented in Figure 14 and that the comprehensive modelling can be found in a technical report [PR06].

In Figure 7 we relate the Korrigan metamodel on which KADL is based with the ADL metamodel we have presented in the introduction.

In our model, both Components and Connectors are represented as *Views*. Both can be simple (using *Integration Views*) or composite (using Composition Views). Architectures are composite components. Ports and Roles correspond to Event Ports. Among the different kinds of *Formulas*, the Indexed Formulas refer to Events of identified subcomponents. These events correspond to the Event Ports of the components. The bindings between component ports and connector roles is achieved through the Indexed Formulas part of composites.

3.1 Interface Description Language

Components, both primitive and composite ones are described using boxes with well-defined interfaces. Their syntax is given in Figure 8.

Component interfaces may be generic on (possibly constrained) data values (*i.e.*, constants, *e.g.*, N,M:Natural {1<N<M}, Fig. 17), data types (*e.g.*, MSG:Sort, Fig. 10), event ports (*e.g.*, G, Fig. 15) and component types (*e.g.*, Server, Client, Fig. 15). As in UML, this is denoted by dashed boxes in the top corners of the components (top left for event ports and component types; top right for data values and data types). In conjunction with genericity on data

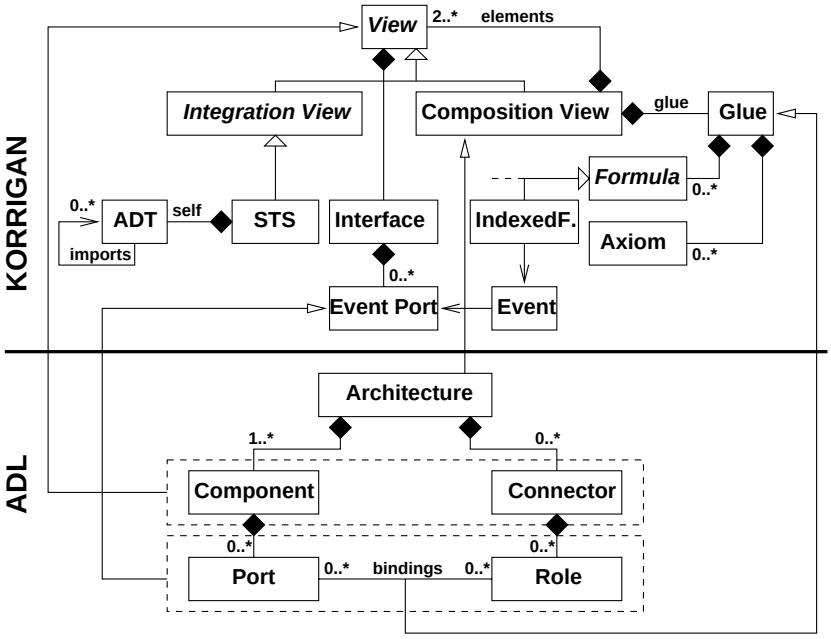


Figure 7: Relation Between Metamodels (UML notation)

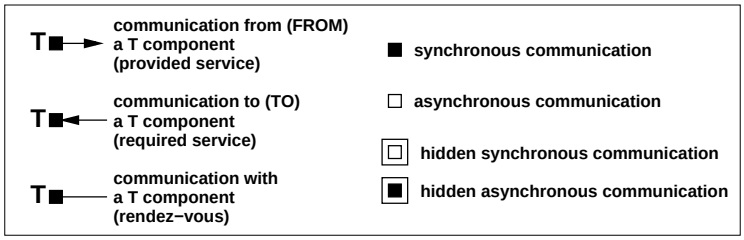


Figure 8: Notation for Interfaces

types and component types, genericity on event ports yields the expressiveness of KADL to express patterns. Patterns may be used in KADL to describe general or common architectures of systems. More information on patterns in KADL will be presented in Section 3.4 (see Fig. 15 and Fig. 16 for example).

Example 6 Interface. The interface of the Till component is given in Figure 9. This component is generic on two instances of a MsgConnection component type. This constrains it to be glued, as far as the send and rec event ports are concerned, with an MsgConnection component or any component that inherits from it (DropMsgConnection, Fig. 10).

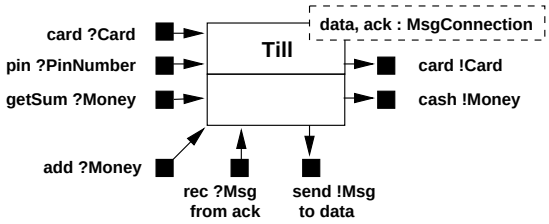


Figure 9: Till Interface

3.2 Primitive Components

Like any other architectural language, KADL provides the designer with first class components. However, in KADL, several kinds of contracts can be associated to components. Both the functional properties (operations) and the dynamic properties (protocols) of these components can be specified. Moreover, an integrated semantics is provided through the use of integration views, namely STS, and their operational semantics which enables one to ensure the consistency between both aspects using verification techniques developed in Section 4.

Example 7 Primitive Components. We have demonstrated the complete set (static and behavioural models) of KADL primitive component notations on the Till component. Its interface has been given in Figure 9, its static model in Figures 4 and 5, and its behavioural model in Figure 6. To be comprehensive, we also give the behavioural models of all the other components used in the global architecture, Figure 14, *i.e.* the connection, BankInterface and DataBase components. Connection components (Fig. 10) are used to model links between tills and the bank interface. Here, we rely on inheritance (UML-like arrow) to specify the relation between links with (possible) failures, DropMsgConnection, and links without message loss, MsgConnection. Failure is modelled using a special down port in DropMsgConnection (right-hand part of Fig. 10). This method may also be used to take into account the dynamic creation or deletion of components.

The behavioural models of the last two components, BankInterface and DataBase, are given in Figure 11. Altogether, they proceed as follows. The tills send withdrawal information to the bank interface they are connected to. This corresponds to an Info type made up of a client identifier and a requested amount of money. The bank interface then passes it to the central database, together with its own identification (which corresponds to a line number). This global information is stored by the database using the lock action. The database then checks if the client is known and if he has enough money on his account. The result of this is then sent back to the till via the bank interface. The information stored is unstored after the database has finished using it.

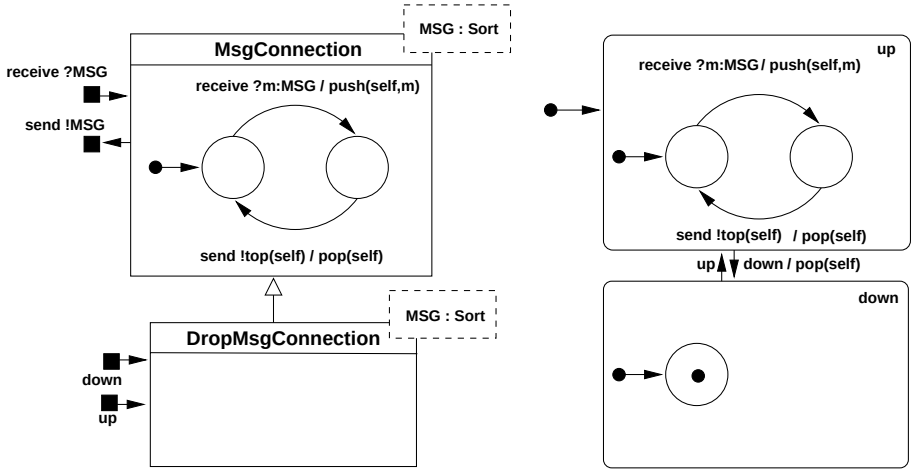


Figure 10: The MsgConnection and DropMsgConnection Behavioural Models

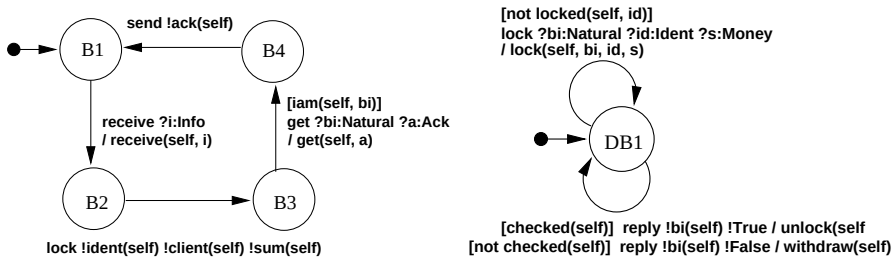


Figure 11: The BankInterface (left) and DataBase (right) Behavioural Models

3.3 Composite Components

Composition diagrams (Fig. 12) are used to represent the component structure of composition views. We reuse the UML class diagram notation with aggregation relations (white diamonds) to denote the concurrent composition of subcomponents into a composite. This has been chosen since the subcomponents of a composite, and more generally the components of an architecture, usually have independent life-cycles. As presented earlier on, we also reuse UML notations for templates/genericity (dashed boxes) and for inheritance (white arrows). We use the usual UML roles on aggregation relations to identify components and extend this notation using our range operator. A component interface may be

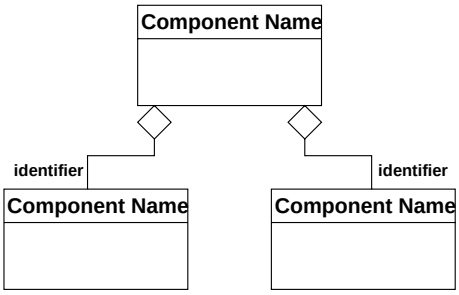


Figure 12: Composition Diagrams

associated with a composition by exporting some events of its subcomponents. Hence, composites are components too, and genericity and pattern issues apply also to them.

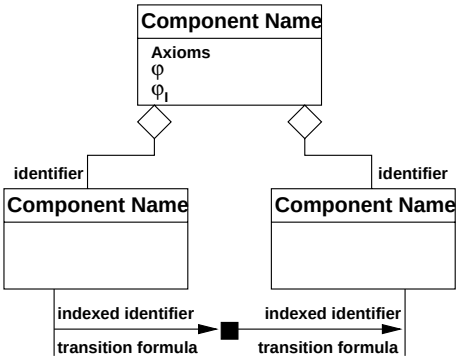


Figure 13: Communication Diagrams

Communication diagrams are composition diagrams complemented with glue rules (Fig. 13). The axioms (Ax_Θ) and the state formulas ($\overline{\psi}$ and $\overline{\psi_{\mathrm{I}}}$) are put in the composite component as they denote invariants of the composite.

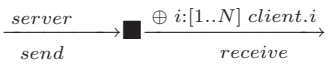
Composition-oriented transition formulas ($\overline{\lambda}$) are represented as follows:

- a communication interface symbol is used to give an information on the communication type;
- lines (or arrows when the communication is directional) between the interface symbol and the components denote to which components the (indexed) subparts of the formula applies;

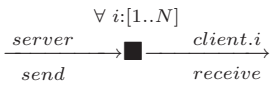
– the identification parts are put above the lines and the formula parts below.

For example, a formula such as $c_1.a_1! \Leftrightarrow c_2.a_2?$ would be represented with an arrow from component c_1, an arrow to component c_2, c_1 and c_2 above the lines, and a_1 and a_2 below:

If a range operator is used, it is also put above the line, on the side of the component to which it applies (*e.g.*, with $server.send! \Leftrightarrow \oplus i : [1..N](client.i.receive?)$ a $\oplus$ would be put on the $client.i$ side)



or above the communication interface symbol if it applies to the whole formula (*e.g.*, with $\forall i : [1..N](server.send! \Leftrightarrow client.i.receive?)$ a $\forall$ would be put above the communication interface symbol).



The graphical representation of modal logic formulas is still an open issue due to the expressiveness of these logics. Some works have addressed this issue for specific logics, *e.g.*, [DKM+94, MRK+97]. A simple solution leads to architectural connectors corresponding to the logic propositional connectors. In our approach we restrict this to the $\Leftrightarrow$ logic connector, which is however the most useful one to represent communication patterns, as demonstrated in 2.4.3. The Reo [Arb04] coordination language adopts a complementary approach: coordination is achieved from a fixed set of basic coordinators, and a graphical notation is defined for each one.

The notion of configurations is also dealt with composite components in KADL as demonstrated in the following example.

Example 8 System Architecture. The system architecture of the till system is given in Figure 14. The till system is made up of N TillLines (*i.e.*, a till and its two communication lines) and the bank with its bank interfaces and database. The range operator is used both to identify components (*e.g.*, the TillLines and the BankInterfaces) and to denote glue.

3.4 Connectors

There is no explicit connector in KADL which rather defines a powerful means to glue things altogether, enabling specific connectors to be defined as particular

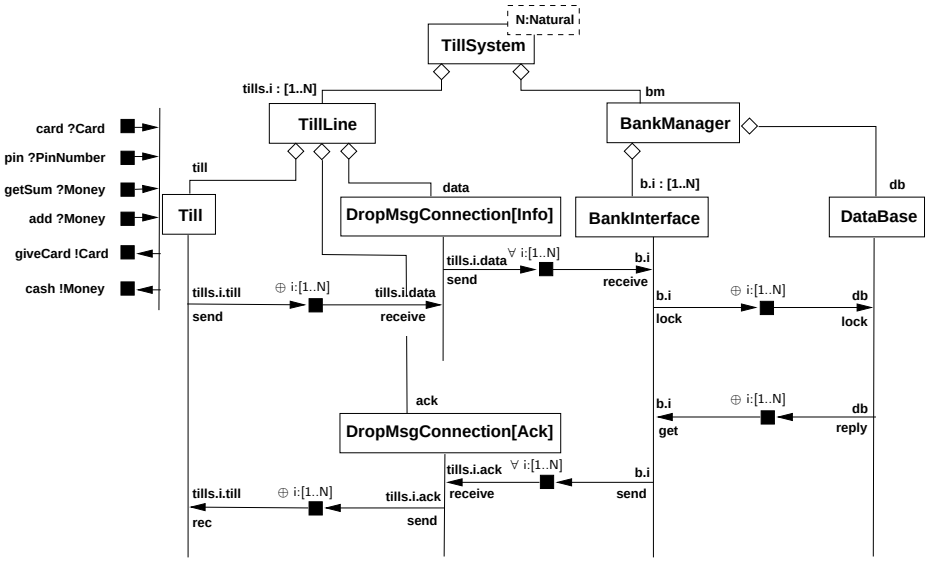


Figure 14: System Architecture Communication Diagram

(generic) components or component patterns. This treatment of connectors is compatible with remarks from [MT00] which state that connectors are particular kinds of components used to model interactions which, however, may not correspond to a compilation unit in the implemented system. To complete this remark we may quote [AG97]: "*explicit connectors are better for intuitive expression of architecture, implicit connectors as components provide a simpler formal semantics*".

A connector may then be seen as a generic pattern resulting from an abstraction over the subcomponents of an architecture. As an example, let us consider a centralized client-server architecture, with a star topology, where the clients may additionally communicate altogether as in a ring network (Fig. 15).

This architecture is made up of a Server and several concurrent Clients. The clients communicate with their two neighbours using their NEXT and PREV ports. The server communicates with a client using G and H ports. The resulting StarRing composite component is generic on the number of clients (N), on the Server and the Client component types, and on the G, H, NEXT, and PREV ports. The Server and the Clients subcomponents are also generic on, respectively, G and {H, NEXT, PREV}.

Instead of requiring a rigid encoding of the communications in the state behaviour of the composition subcomponents, KADL relies on an external (exogenous) glue, put in the composition models. For example, in the $\overline{\lambda}$ part, one

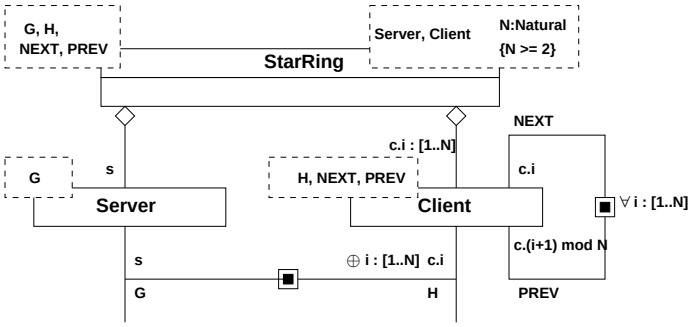


Figure 15: StarRing Connector/Pattern using a Communication Diagram

may write $(s.G \Leftrightarrow c.i.H)$ to synchronize G of s:Server and H of c.i:Client. To formally express that communications are exclusive between the server and its clients (*i.e.*, one client treated at a time), this formula can be refined using the $\oplus$ operator: $s.G \Leftrightarrow \oplus i : [1..N]\ (c.i.H)$. As explained before, these communication constraints may be graphically represented, as in Figure 15, by a line joining the two components with a $\oplus$ quantifier. The $\forall$ quantifier, in the same figure, denotes that the inter-client communication represented here is valid for any client.

The generic pattern of Figure 15 allows one to instantiate StarRing connectors. One first has to give a concrete server component and several concrete clients components, and then instantiate the different port parameters. This example demonstrates that connectors naturally arise in KADL thanks to its first class components and its external glue.

3.5 Dynamic Architectures

Architectural descriptions in KADL are inherently dynamic in the sense that, taking the behavioural aspects of components into account, one may get an operational semantics from the composition of the architectural components using the definitions in Section 2. Dynamic architectures also means to adapt the architecture to changes in a system. A demonstrative example is the Faulty-Tolerant Client-Server (FTCS) [AG97]. This is a classic client-server architecture with an auxiliary server that can be used whenever the main server goes down. A KADL specification of FTCS is given in Figure 16.

This example is based on the following communication scheme. Whenever the main server is on (*i.e.*, the formula $main.@on$ yields) then the server and the clients may synchronize on G/H. If the main server is off while the auxiliary server is on, then the same can be achieved using the auxiliary server in place

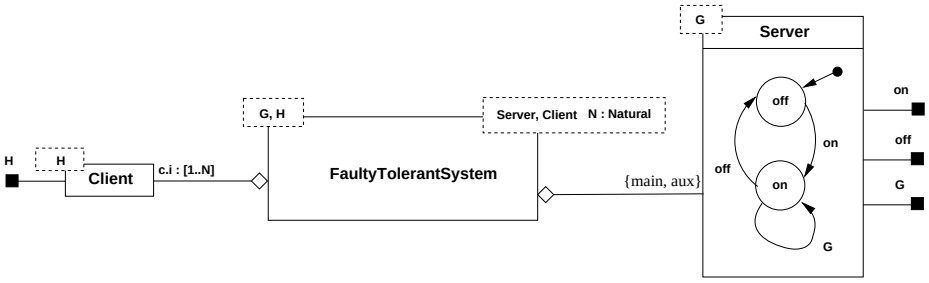


Figure 16: Faulty-Tolerant Client-Server

of the main server. The corresponding glue is:

$$(main.@on \wedge (main.G \Leftrightarrow \oplus i : [1..N]c.i.H)) \vee$$
$$(main.@off \wedge aux.@on \wedge (aux.G \Leftrightarrow \oplus i : [1..N]c.i.H))$$

Here, most of the basic client-server pattern was kept, as it was possible to extend it into a FTCS using the glue. However, dynamic reconfiguration often yields more complex situations. Dynamic reconfiguration may indeed mean two (non exclusive) things:

1. dynamic modification of the communications, as in the FTCS example;
2. dynamic modification of the architectural set of components.

To demonstrate the ability of KADL to support also the second case, let us take into account a modified version of our case study. The tills may now be up or down, we reuse the principles of Figure 10 for this. The new architecture is given in Figure 17.

N-1 tills are connected by dedicated TillLines, one each, as before. The N^{th} line is now used to connect a Multiplexer and additional Tills (from N to M). The interface of the multiplexer has to be the same as a till line, yet its dynamic behaviour is different. The glue in the multiplexer has to connect any of its related tills with the b.i.N bank interface. This link must be valid during a complete cycle of send and receive to avoid the possible interleaving of communications between the different tills connected to the multiplexer and the rest of the system. Such treatment of cycles may be dealt with in KADL using the modal $[\overline{\lambda}]\overline{\psi}$ operator and the binding operator on states, $\overset{@}{\exists}$. The glue of the multiplexer for one till (the j^{th}) and without failure is $\overline{\Phi}_j$:

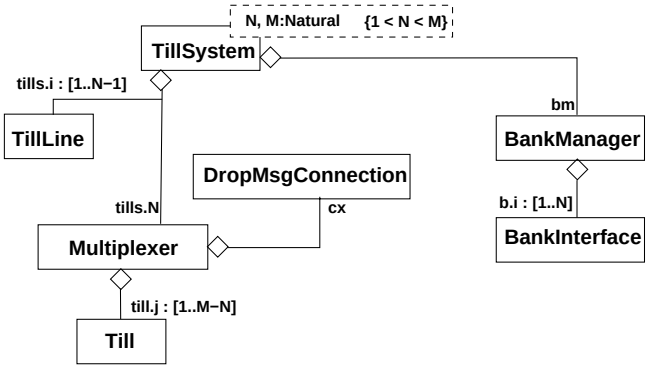


Figure 17: Multiplexer for the Till System

$$\overline{\phi}_1 = self.till.j.send! \Leftrightarrow self.cx.receive?$$
$$\overline{\phi}_2 = self.cx.send! \Leftrightarrow self.send!$$
$$\overline{\phi}_3 = self.receive? \Leftrightarrow self.cx.receive?$$
$$\overline{\phi}_4 = self.cx.send! \Leftrightarrow self.till.j.rec?$$
$$\overline{\Phi}_j = \overset{@}{\exists}\, \overline{s}.[\overline{\phi}_1][\overline{\phi}_2][\overline{\phi}_3][\overline{\phi}_4]@\overline{s}$$

where $\overset{@}{\exists}\, \overline{s}.[\overline{\phi}_1][\overline{\phi}_2][\overline{\phi}_3][\overline{\phi}_4]@\overline{s}$ denotes a cycle starting from a given state (denoted by $\overline{s}$), satisfying in sequence $\overline{\phi}_1 \ldots \overline{\phi}_4$, and ending in the $\overline{s}$ state.

Formula $\overline{\Phi}_j$ may be extended to take all the multiplexed tills into account using the exclusive disjunction operator: $\overline{\Phi}_{\mathsf{Multiplexer}} = \oplus j : [1..M-N].\overline{\Phi}_j$.

4 Architecture Verification

In order to promote the use of formal methods in the industrial world, we agree with [Bro85]: *most important properties of specifications methods are not only the underlying theoretical concepts but more pragmatic issues such as readability, tractability, support for structuring, possibilities of visual aids and machine support*. The need for machine support, *i.e.*, tools, fully yields for formal architectural languages [MT00]. In this Section we will briefly present tool-equipped techniques that we have developed which can be applied to KADL architectures. We will first present the finite domain ones, *i.e.*, techniques which arbitrarily restrict the domain of the data types in order to obtain finite LTS from STS. We will then present new techniques we have developed which rely on the system architecture and on bounded analysis to obtain an abstraction of the system which can be verified.

4.1 Finite Domain Techniques

In [CPR01b], we have proposed a framework dedicated to Korrigan which followed two main principles: openness and extensibility. According to these principles, it provided a library for STS and supported model transformation into other mixed formalisms (Larch, LOTOS and SDL) with the objective to take advantage of their dedicated verification tools (model checkers or theorem provers). As the set of possible target model languages evolve, the framework was based on a class library reifying the different models taken into account. Transformations were implemented as methods within these classes. This design made the framework extensible.

Our objective was also to provide tools which could be used in external formal frameworks. Therefore, we had defined the CLAP (Class Library for Automata in Python) library [CPR01b] which enabled one to perform (a)synchronous composition of any state-transition based models. CLAP served as a basis for xCLAP [APS07], a tool dedicated to the animation of UML state diagrams extended with formal data types (either Larch, Z or B). CLAP has been recently reimplemented and extended as a Java plug-in for Eclipse, ETS [Poi05]. ETS is, as far as we know, the only tool for state-transition composition which (i) enables the extension of components with any user-defined feature associated to either states or transitions, (ii) supports an expressive external composition description language for models (glue) and (iii) keeps the structuring on the models resulting from the composition. ETS is used in Adaptor, a tool dedicated to model-based software adaptation [CPS06], in order to obtain the models of adaptors which solve behavioural mismatch between components.

Another important feature of the framework presented in [CPR01b] was the ability to generate code in a concurrent object-oriented language, Active Java, from the specifications. This was based on a four steps mechanism, using a hierarchical approach where control nodes enforced an asynchronous model of concurrency and a purely synchronous communication between components. The code generation has been recently improved in [PNPR05] using pure Java. This implementation relies on controllers which encapsulate protocols and channels devoted to (possibly remote) communications between components. This enables both synchronous and asynchronous communication and avoids the need for centralizing control nodes.

4.2 Symbolic Analysis Techniques

The main drawback of the framework presented in [CPR01b] is that it relies on model transformation into mixed formalisms whose tools impose to restrict the domains of variables in order to get finite transition systems (LTS) from the STS specifications. In cases where a lot of, or complex data types are used, this may

prevent verification to be performed on the models (see 4.3, below). Therefore, since [CPR01b], we have developed symbolic verification techniques which take advantage of the architectural descriptions and the symbolic description level of the dynamic models.

We first worked on boundedness procedures for message queues (mailboxes) of components described using STS. This is an important property of component-based systems as mailbox unboundedness may cause message loss and service denial. At the model level, unconstrained mailboxes are also a reason for the models state explosion when verifying them. We have proposed semi-decidable procedures for this in [MPR04] using different mailbox protocols, namely FIFO (First In First Out) and DICO, an hash-table based abstraction of FIFO. This proposal has then been extended with the notions of *bounded analysis* and *bounded projection* [PRS06]. Bounded analysis tests the boundedness of possibly infinite systems or part of them (in presence of composition and communication) and generates finite simulations over which verification can be performed. Bounded projection is an approach which can make bounded analysis more efficient. The idea is to select a subset of the data used in the behavioural models (*e.g.*, using properties of interest and slicing) and then do a partial evaluation of STS using it. The computation of configuration graphs is adjusted to evaluate only guards and actions related to the selected data. One can then analyse parts of an STS which can be bounded and then build an abstraction of it.

These symbolic analysis techniques have been implemented in SyCLAP, an extension of CLAP (about 4000 lines of Python). They have been successfully applied to several benchmarks, including different distributed systems protocols (several versions of the bakery protocol, the slip protocol, resource allocator protocols), a component based flight reservation system and the example used in this article.

We have also been interested in theorem proving related techniques for the verification of STS, focusing on the development of automatic proof strategies depending on the properties form, and on the interaction between theorem proving and model checking. In [AR02, NPR04] we have shown that the PVS theorem prover could be used to conduct proofs on STS models. These works have been extended in [Roy04] where a temporal logic for STS, CTL*Data, and dedicated PVS proof techniques, have been defined. Deadlock freedom, but also properties involving data types have been successfully proved in this context.

4.3 Application to the Example

In this Section we demonstrate verification on our case-study. Comprehensive descriptions and verifications of the models, together with more discussion on the limits of the finite domain techniques, are described in a technical report [PR06].

An important remark is that the verification of our example with a representative finite domains tool, namely CADP with LOTOS specifications, has failed due to the high inter-relation between the behavioural and the static (data types) aspects in the case study.

Hypotheses and verification technique. In order to illustrate the use of SyCLAP, we work under the following hypotheses. Clients may type in either correct or wrong PIN codes. Different (any) amounts can be asked to be withdrawn. We consider the withdrawal daily limit for an account i as a parameter MAXi. Closing systems by modelling (restricting the behaviours of) its external environment is a way to reduce the verification complexity. Yet, this prevents full-fledged verification. Here the system is not a closed one, *i.e.*, we do not enforce a given client behaviour: the client may perform actions in any order and with any values, see Figure 18).

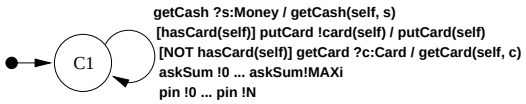


Figure 18: Unconstrained Client Model

Then, a global STS for the system can be built from the component STSs, several client STSs (depending on N, the number of tills in the system) and the architectural description (which defines correspondences between component events), and afterwards corresponding configuration graphs can be derived from this global STS. Information on these graphs, for different system parameters, are given in tables, *e.g.*, Table 1 (due to lack of place, these graphs have not been given here because of their size). Sizes are indicated as couples (S,T) where S is the number of states and T the number of transitions. Variables and constants not specified in the tables are kept unbound; this is a major difference with finite domains tools which would require one to bound all data type domains.

N	#account	max Till amount	MAX1	MAX2	size
2	0	1	0	0	(324, 1224)
2	0	1	1	0	(1692, 6724)
2	0	1	1	1	(8872, 36992)
2	1	1	1	1	(15912, 67176)

Table 1: Verification – Configuration Graphs (Part I)

In Table 1 one can observe that even with only two tills and a single bank account (remember that accounts can be shared unless otherwise specified), the size of the configuration graphs can grow quite big. This is due to the numerous other variables and constants that have an effect on the system, and this makes important the use of symbolic techniques.

Model refinement from verification. As explained before, we chose not to close the system by restricting the clients behaviours. However, symbolic verification on the global system using SyCLAP allowed us to find out several problems. We discovered for example that after swallowCard takes place, the passing of time (clock) may become the only possible transition of the system. Moreover, we noticed that the set of states where only clock is possible corresponded exactly to the targets of a swallowCard event. This meant that the two were closely related. The problem was due to the fact that the tills keep cards after three successive wrong PIN identifications. The system could be corrected with a more advanced model, where a client action was added to get the card back.

Safety properties and bounded projection. Another experiment was to check if the system ensured an exclusive access to bank accounts, which was related to the possibility of having several clients interacting at the same time with different tills in the system. A counter-example would have been that two clients, with the same account number, would withdraw at the same time using two distinct tills. There, we have applied our abstraction techniques as follows to avoid state explosion.

The parts corresponding to the database and bank interfaces have been checked, while the remaining of the system was abstracted and a dedicated component was devoted to the simulation of the tills, the clients and the communication links. The database contains both information on client accounts (account identification and money amount, *i.e.*, Ident x Money) and information related to communications (as explained in 3.2: communication line, account information, and withdrawal request, *i.e.*, Natural x Ident x Money). The overall information yield by the database is then List[Ident x Money] x List[Natural x Ident x Money], with constraints that make it equivalent to List[Natural x Ident x Money]. Using this information, a bounded projection decomposition over only List[Natural x Ident], the one interesting for our exclusive access property, could be achieved on the DataBase component STS. Then, building projection and configuration graphs with SyCLAP, it was possible to prove that the desired exclusive access property yield, *e.g.*, for N=2, #account=10, MAX=3 and N=3, #account=4, MAX=2 (see Tab. 2 for information on the related configuration graphs, note again that unless specified, variables and constants are left unbound).

Other safety properties have also been verified in the same way, *e.g.*, that the PIN counter is always equal to three after a swallowCard, and that both database and till amounts are always greater or equal than zero.

N	MAX	#account	size
2	2	1	(52, 118)
2	2	2	(193, 564)
2	2	10	(4561, 24580)
2	3	1	(177, 484)
2	3	2	(713, 2568)
2	3	10	(17961, 145960)
3	2	1	(309, 966)
3	2	2	(2351, 9978)
3	2	4	(19461, 107292)
3	3	1	(1895, 7290)

Table 2: Verification – Configuration Graphs (Part II)

Manual abstraction over STS. Note that other abstraction techniques such as [CGL94, DGG97, BLO98, MV98] could be used in our context. However, this would require complex manual transformations of the model and the properties. Meanwhile, some abstractions are quite simple to perform on our STS, either on the behavioural part or the data part.

As an example, we wanted to check that an existing card is either owned by the proper client or by the till the client is connected to. This property has been proven by abstracting the data of the system into the card identity type which also corresponds to the clients ids. The global product has been computed for N=1, 2 and 3 without having to choose effective values for the other parameters. The obtained corresponding configuration graphs were bounded (see Tab. 3) and the property could be checked.

N	size
1	(24, 56)
2	(576, 2688)
3	(13824, 96768)

Table 3: Verification – Configuration Graphs (Part III)

The abstractions of the components STSs were automatic, but the abstractions on the ADT have been achieved manually. Automation is a perspective.

5 Related Work and Models

Numerous works have addressed the formal description and analysis of components and architectures. Most of them proceed by adapting in an ADL framework different process algebras (*e.g.*, CSP and the FDR tool for Wright [AG97], π-calculus for Darwin [MK96], FSP and the LTSA tool for [MKG99, FUMK03], EMPA for PADL [BCD01] and AEmilia [BBS02]). Graphical representations for architectures have been initially supported by extensions of semi-formal notations such as UML 1.x [MRRR02, SR98]. They are now fully supported by UML 2.0 [OMG05]. In this section we compare the KADL ADL with representatives of these two families, namely Wright for the formal ADLs and UML for the graphical notations that support ADL descriptions. To end, we will relate KADL with coordination languages as they share common features.

5.1 Formal ADLs

Wright [AG97] is a representative of process algebra based ADLs. It is a formal ADL with first class components and connectors which can be seen as relook of CSP, since its syntax and semantics are related to this process algebra and since the FDR tool is used to verify Wright specifications. Architectures in Wright are made up of three parts: type definitions, configurations and bindings. The first part is used to define both component and connector types. A component is given as a set of ports and a behaviour over these ports. A connector defines a set of roles and a glue specification. A connector role describes the expected local behaviour of the component interacting at this connector and the connector glue then describes how the local activities of the different roles are coordinated altogether. The second part of a Wright specification describes an architectural configuration as a set of instances of component types and connector types. Finally, the third part describes how the component and connector instances are connected to define the complete system. Wright supports dynamic reconfiguration. Specific events denote when reconfiguration is permitted and are used in a separate view of the architecture. The configuration program (configuror) describes how these events trigger reconfigurations.

The semantics of the Wright constructions is defined by translation into CSP. This enables one to take advantage of CSP model-checking and behavioural refinement techniques. Wright allows one to check for connector, configuror, and attachment consistency using mainly techniques to prove deadlock freedom and behavioural refinement.

Data types are an important means to define a relation between the dynamic interface (events) and the functional interface (operations) of components. In KADL, not only simple dynamic properties but also properties taking into account semantics information (related to the ADT) are therefore supported, as

demonstrated in 4.3. Meanwhile, verification is limited in Wright by state explosion in case of value-passing or value encapsulation. Wright therefore supports only very simple, bounded, data types. This is an important limitation of all ADLs based on process algebras. The support of full data types is a major difference between Wright and KADL. More generally KADL supports STS related models and is not limited to LTS finite state models such as Wright.

In Wright, connectors explicit the protocol of the connected components and describe, through the glue, the expected global behaviour of the system using process algebraic expressions. KADL is more abstract since its glue denotes modal logic properties of the global system and not directly the global dynamic behaviour, even if the semantics enables one to obtain it. KADL improves also readability thanks to graphic notations for behaviours, while Wright process algebraic notation may be less legible. Finally, Wright has no support for n-ary composition.

5.2 Graphical Semi-Formal Notations

As seen earlier on, KADL is supported by an UML-inspired graphical notation. When possible, we suggest to reuse and extend such common software engineering notations in the design of new formal languages. This paves the way for their integration in model driven engineering (MDE) and on a wider scale it should help in their acceptation outside the academic community.

A major problem with UML 1.x was its lack of support for components and for the definition of clear architectures of concurrent systems [MM98, CPR01a, Boc04]. Extensions of UML such as UML-RT [SR98] have partly addressed the same issues than KADL: architectural design, dynamic and functional interfaces of components, and reusability. While UML-RT is a purely design level notation, KADL is concerned about formal specification and verification issues. There are also some other differences, mainly at the communication level, but the major one is that, to the contrary of UML-RT, KADL enables one to specify both active, reactive and proactive systems in a uniform way.

UML 2.0 extends, cleans up and clarifies UML 1.x on several points [PD03, Boc04]. We will focus here on architecture and scalability related ones. UML 2.0 supports a notion of component with ports, a first-class concept that denotes instantiable connections. The UML 1.x interface concept is extended to deal with the required and provided aspects of component interfaces. Protocol state machines, a restricted form of UML state diagrams, can be used to specify service invocation sequences in interfaces. Structured classes allow the definition of hierarchical structures. All these elements are valuable, but mainly syntactic, improvements which increase the ability and utility of UML to deal with architecture and scalability. However, due to its expressiveness, UML still lacks means to check consistency of dynamic behaviours and data types.

Our concerns about methods and graphical notations for formal languages are close to [RL97, CR99] ones. However, we think we can reuse UML notations, or partly extend them using stereotypes or profiles, rather than defining new notations from scratch. Our notations are also more expressive and abstract than [RL97] as far as communication issues are concerned. Our approach is dual to the theoretical approaches that try to formalize the UML [APS07].

5.3 Coordination Languages

In comparison with more usual languages in which the interaction part of compositions (*i.e.*, communication, synchronizing) is embedded within the computation part, coordination languages [PA98] promote separation of concerns hence the definition of interaction as a first-class entity, described separately from computation. Coordination languages are split into two categories. Data driven languages propose expressive but low level communication mechanisms based on shared data spaces, such as Linda tuple spaces and their Java implementation, javaspaces. On the contrary, event driven languages such as Reo [Arb04] promote more abstract coordination patterns based on events corresponding to the coordinated entities input and output ports.

Our proposal is clearly related to this second category which is, in our opinion, more adequate for the design process than data spaces. The KADL glue, used separately from the definition of components, can be seen as an expressive way of modelling coordination patterns. Moreover, taking full formal data types into account, we extend event based coordination to interactions involving data. Using STS in place of more usual LTS, we also have a very abstract description means for behavioural interface description languages and protocols which are the support for coordination.

6 Conclusion

Architectural Description Languages (ADL) promote abstraction and separation of concerns in Component Based Software Engineering. They focus on composition and interaction aspects of systems supporting the component design and deployment process. They are also the ground on which formal methods can be applied to analyse component architectures, coordinate components and, if required, adapt them. As component based systems get more complex, new requirements arise for ADLs and components interfaces description languages. Interface descriptions must be available for operations, behaviours, semantics and quality of service (QoS); and formal techniques must address, when possible, the specification and verification at these four description levels.

In this article, we addressed both the operation, behavioural and semantics levels through the definition of KADL, a formal ADL based on ideas from the

Korrigan specification language. This ADL enables one to describe component architectures formally, at a good abstraction level, using expressive interaction structuring mechanisms based on modal logic. Both operations, their semantics and the behaviours of components are taken into account thanks to the use of Symbolic Transition Systems (STS), a mixed specification model supporting the integration of fully formal data types into behaviours. We also addressed the verification of component architectures through analysis techniques dedicated to the use of STS in the context of interacting components. This enables one to avoid the well-known state explosion problem arising when verifying behavioural protocols integrating data types into lower level formal models such as Labelled Transition Systems (LTS).

Perspectives concern the integration of KADL in the MDE process trough model transformations from design level notations such as UML 2.0 to KADL and from KADL to component implementation languages such as Fractal which, thanks to its hierarchical structure, may more easily support the transformation process. We are also working on the integration of (temporal or resource based) QoS properties in architectures and the extension of our verification and adaptation techniques to this level.

Acknowledgements

We would like to thank the anonymous reviewers for their numerous remarks which have helped us to improve the article.

References

[ABP04] M. Aiguier, F. Barbier, and P. Poizat. A Logic with Temporal Glue for Mixed Specifications. In *Foundations of Coordination Languages and Software Architectures (FOCLASA'03)*, volume 97 of *Electronic Notes in Theoretical Computer Science*, pages 155–174, 2004.

[AG97] R. J. Allen and D. Garlan. A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, 1997.

[AKBK99] E. Astesiano, B. Krieg-Brückner, and H.-J. Kreowski, editors. *Algebraic Foundation of Systems Specification*. IFIP State-of-the-Art Reports. Springer-Verlag, 1999.

[APS07] C. Attiogbé, P. Poizat, and G. Salaün. A Formal and Tool-Equipped Approach for the Integration of State Diagrams and Formal Datatypes. *IEEE Transactions on Software Engineering*, 33(2), February 2007. to appear.

[AR02] M. Allemand and J.-C. Royer. Mixed Formal Specification with PVS. In *Workshop on Formal Methods for Parallel Programming (FMPPTA'2002) at the International Parallel and Distributed Processing Symposium (IPDPS)*, 2002.

[Arb04] F. Arbab. Reo: a channel-based coordination model for component composition. *Mathematical Structures in Computer Science*, 14(3):329–366, 2004.

[BBS02] S. Balsamo, M. Bernardo, and M. Simeoni. Combining Stochastic Process Algebras and Queueing Networks for Software Architecture Analysis. In *International Workshop on Software and Performance (WOSP'2002)*, pages 190–202, 2002.

[BCD01] M. Bernardo, P. Ciancarini, and L. Donatiello. Detecting Architectural Mismatches in Process Algebraic Description of Software Systems. In *Working IEEE/IFIP Conference on Software Architecture (WICSA'2001)*, pages 77–86, 2001.

[Béz05] J. Bézivin. On the Unification Power of Models. *Software and System Modeling*, 4(2):171–188, 2005.

[BI03] M. Bernardo and P. Inverardi, editors. *Formal Methods for Software Architectures*, volume 2804 of *Lecture Notes in Computer Science*. Springer-Verlag, 2003.

[BLO98] S. Bensalem, Y. Lakhnech, and S. Owre. Computing Abstractions of Infinite State Systems Compositionally and Automatically. In *Computer-Aided Verification (CAV'98)*, volume 1427 of *Lecture Notes in Computer Science*, pages 319–331. Springer-Verlag, 1998.

[Boc04] C. Bock. UML2 Composition Model. *Journal of Object Technology*, 3(10):47–73, 2004.

[Bro85] M. Broy. Specification and Top Down Design of Distributed Systems. In *International Joint Conference on Theory and Practice of Software Development (TAPSOFT'85)*, volume 185 of *Lecture Notes in Computer Science*, pages 4–28. Springer-Verlag, 1985.

[CGL94] E. M. Clarke, O. Grumberg, and D. E. Long. Model-Checking and Abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, 1994.

[CMP06] C. Canal, J. M. Murillo, and P. Poizat. Software Adaptation. *L'Objet*, 12(1):9–31, 2006. Special Issue on Coordination and Adaptation Techniques for Software Entities.

[CMS02] M. Calder, S. Maharaj, and C. Shankland. A Modal Logic for Full LOTOS Based on Symbolic Transition Systems. *The Computer Journal*, 45(1):55–61, 2002.

[CPR99] C. Choppy, P. Poizat, and J.-C. Royer. From Informal Requirements to COOP: a Concurrent Automata Approach. In *Formal Methods Conference (FM'99)*, volume 1709 of *Lecture Notes in Computer Science*, pages 939–962. Springer-Verlag, 1999.

[CPR00] C. Choppy, P. Poizat, and J.-C. Royer. A Global Semantics for Views. In *Algebraic Methodology And Software Technology (AMAST'00)*, volume 1816 of *Lecture Notes in Computer Science*, pages 165–180. Springer-Verlag, 2000.

[CPR01a] C. Choppy, P. Poizat, and J.-C. Royer. Specification of Mixed Systems in Korrigan with the Support of a UML-Inspired Graphical Notation. In *Fundamental Approaches to Software Engineering (FASE 2001)*, volume 2029 of *Lecture Notes in Computer Science*, pages 124–139. Springer-Verlag, 2001.

[CPR01b] C. Choppy, P. Poizat, and J.-C. Royer. The Korrigan Environment. *Journal of Universal Computer Science*, 7(1):19–36, 2001. Special issue: Tools for System Design and Verification.

[CPS06] C. Canal, P. Poizat, and G. Salaün. Synchronizing Behavioural Mismatch in Software Composition. In *International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS'06)*, volume 4037 of *Lecture Notes in Computer Science*, pages 63–77. Springer-Verlag, 2006.

[CR99] E. Coscia and G. Reggio. JTN: A Java-Targeted Graphic Formal Notation for Reactive and Concurrent Systems. In *Fundamental Approaches to Software Engineering (FASE'99)*, volume 1577 of *Lecture Notes in Computer*

Science, pages 77–97. Springer-Verlag, 1999.

[CVZ06] I. Cerná, P. Vareková, and B. Zimmerova. Component Substitutability via Equivalencies of Component-Interaction Automata. In *International Workshop on Formal Aspects of Component Software*, Electronic Notes in Theoretical Computer Science. Elsevier, 2006.

[DGG97] D. Dams, R. Gerth, and O. Grumberg. Abstract Interpretation of Reactive Systems. *ACM Transactions on Programming Languages and Systems*, 19(2):253–291, 1997.

[DKM+94] L. K. Dillon, G. Kutty, L. E. Moser, P. M. Melliar-Smith, and Y. S. Ramakrishna. A Graphical Interval Logic for Specifying Concurrent Systems. *ACM Transactions on Software Engineering and Methodology*, 3(2):131–165, 1994.

[DMY02] A. David, M. O. Möller, and W. Yi. Formal Verification of UML Statecharts with Real-Time Extensions. In *International Conference on Fundamental Approaches to Software Engineering (FASE'02)*, volume 2306 of *Lecture Notes in Computer Science*, pages 218–232. Springer-Verlag, 2002.

[DOP00] T. Denvir, J. Oliveira, and N. Plat. The Cash-Point (ATM) 'Problem'. *Formal Aspects of Computing*, 12(4):211–215, 2000.

[FECA05] R. E. Filman, T. Elrad, S. Clarke, and M. Akşit. *Aspect-Oriented Software Development*. Addison-Wesley, 2005.

[Fis97] C. Fischer. CSP-OZ: a combination of Object-Z and CSP. In *2nd IFIP Workshop on Formal Methods for Open Object-Based Distributed Systems (FMOODS'97)*, pages 423–438. Chapman & Hall, 1997.

[FUMK03] H. Foster, S. Uchitel, J. Magee, and J. Kramer. Model-based Verification of Web Service Compositions. In *International Conference on Automated Software Engineering (ASE'2003)*, pages 152–163. IEEE Computer Society, 2003.

[GLM01] H. Garavel, F. Lang, and R. Mateescu. An Overview of CADP 2001. *EASST Newsletter*, 4:13–24, 2001.

[Har87] D. Harel. Statecharts: A Visual Formalism for Complex System. *Science of Computer Programming*, 8(3):231–274, 1987.

[IL01] A. Ingolfsdottir and H. Lin. *A Symbolic Approach to Value-passing Processes*, chapter Handbook of Process Algebra. Elsevier, 2001.

[ISO89] ISO/IEC. LOTOS: A Formal Description Technique based on the Temporal Ordering of Observational Behaviour. ISO/IEC 8807, International Organization for Standardization, 1989.

[IT02] ITU-T. Specification and Description Language (SDL), 2002.

[JJRZ05] B. Jeannet, T. Jéron, V. Rusu, and E. Zinovieva. Symbolic Test Selection Based on Approximate Analysis. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'2005)*, volume 3440 of *Lecture Notes in Computer Science*, pages 349–364. Springer-Verlag, 2005.

[KLM+97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Videira Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *European Conference on Object-Oriented Programming (ECOOP'97)*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242, 1997.

[Lam94] L. Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, 1994.

[Mes92] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[MK96] J. Magee and J. Kramer. Dynamic Structure in Software Architectures. In *ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, pages 3–14, 1996.

[MKG99] J. Magee, J. Kramer, and D. Giannakopoulou. Behaviour Analysis of Software Architectures. In *Working IFIP Conference on Software Architecture*

(WICSA1), volume 140 of *IFIP Conference Proceedings*, pages 35–50, 1999.

[MM98] M. J. McLaughlin and A. Moore. Real-time extensions to UML. *Dr. Dobb's Journal of Software Tools*, 23(12):82, 84, 86–93, 1998.

[MPR04] O. Maréchal, P. Poizat, and J.-C. Royer. Checking Asynchronously Communicating Components using Symbolic Transition Systems. In *Internationa Symposium on Distributed Objects and Applications (DOA'04)*, volume 3291 of *Lecture Notes in Computer Science*, pages 1502–1519. Springer-Verlag, 2004.

[MRK+97] L. E. Moser, Y. S. Ramakrishna, G. Kutty, P. M. Melliar-Smith, and L. K. Dillon. A Graphical Environment for the Design of Concurrent Real-Time Systems. *ACM Transactions on Software Engineering and Methodology*, 6(1):31–79, 1997.

[MRRR02] N. Medvidovic, D. S. Rosenblum, D. F. Redmiles, and J. E. Robbins. Modeling Software Architectures in the Unified Modeling Language. *ACM Transactions on Software Engineering and Methodology*, 11(1):2–57, 2002.

[MT00] N. Medvidovic and R. N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, 2000.

[MV98] F. Michel and F. Vernadat. Maîtriser l'explosion combinatoire, réduction du graphe de comportement. *RAIRO, Technique et Science Informatiques*, 17:805–837, 1998.

[Nie95] O. Nierstrasz. Regular types for active objects. In *Object-Oriented Software Composition*, pages 99–121. Prentice Hall, 1995.

[NPR04] J. Noyé, S. Pavel, and J.-C. Royer. A PVS Experiment with Asynchronous Communicating Components. In *Workshop on Algebraic Development Techniques (WADT'2004)*, 2004.

[OMG05] OMG. UML Superstructure Specification, v2.0. Document formal/05-07-04, August 2005.

[OMG06] OMG. CORBA Component Model Specification, v4.0. Document formal/06-04-01, April 2006.

[PA98] G. A. Papadopoulos and F. Arbab. Coordination Models and Languages. In *The Engineering of Large Systems*, volume 46 of *Advances in Computers*, pages 329–400. Academic Press, 1998.

[PD03] B. Powel Douglass. UML 2.0: Incremental Improvements for Scalability and Architecture. Technical report, I-Logix Inc., 2003.

[Pla03] D. S. Platt. *Introducing Microsoft .NET, Third Edition*. Microsoft Press, 2003.

[PNPR05] S. Pavel, J. Noyé, P. Poizat, and J.-C. Royer. A Java Implementation of a Component Model with Explicit Symbolic Protocols. In *International Workshop on Software Composition (SC'05)*, volume 3628 of *Lecture Notes in Computer Science*, pages 115–124. Springer-Verlag, 2005.

[Poi00] P. Poizat. *Korrigan : un formalisme et une méthode pour la spécification formelle et structurée de systèmes mixtes*. PhD thesis, University of Nantes, France, December 2000.

[Poi05] P. Poizat. Eclipse Transition Systems. French National Network for Telecommunications Research (RNRT), STACS Project Deliverable, 2005.

[PR06] P. Poizat and J.-C. Royer. KADL specification of the cash point case study. Technical Report RR 2006-07, IBISC FRE 2873 CNRS, Université d'Évry Val d'Essonne, December 2006.

[PRS04] P. Poizat, J.-C. Royer, and G. Salaün. Formal Methods for Component Description, Coordination and Adaptation. In *International Workshop on Coordination and Adaptation Techniques for Software Entities (WCAT'04) at ECOOP*, pages 89–100, 2004.

[PRS06] P. Poizat, J.-C. Royer, and G. Salaün. Bounded Analysis and Decomposition for Behavioural Descriptions of Components. In *International*

Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS'06), volume 4037 of *Lecture Notes in Computer Science*, pages 33–47. Springer-Verlag, 2006.

[PV02] F. Plasil and S. Visnovsky. Behavior Protocols for Software Components. *IEEE Transactions on Software Engineering*, 28(11):1056–1076, November 2002.

[RL97] G. Reggio and M. Larosa. A Graphic Notation for Formal Specifications of Dynamic Systems. In *Formal Methods Conference (FM'97)*, volume 1313 of *Lecture Notes in Computer Science*, pages 40–61. Springer-Verlag, 1997.

[Roy03] J.-C. Royer. The GAT Approach to Specify Mixed Systems. *Informatica*, 27(1):89–103, 2003.

[Roy04] J.-C. Royer. A Framework for the GAT Temporal Logic. In *International Conference on Intelligent and Adaptive Systems and Software Engineering (IASSE'04)*, 2004.

[Smi97] G. Smith. A Semantic Integration of Object-Z and CSP for the Specification of Concurrent Systems. In *Formal Methods Europe (FME'97)*, volume 1313 of *Lecture Notes in Computer Science*, pages 62–81. Springer-Verlag, 1997.

[Sou96] J.-L. Sourrouille. A framework for the definition of behavior inheritance. *Journal of Object-Oriented Programming*, 9(1):17–21, 1996.

[SR98] B. Selic and J. Rumbaugh. Using UML for Modeling Complex Real-Time Systems. Technical report, Rational Software Corp., 1998.

[Sun03] Sun Microsystems. Java 2 Platform Enterprise Edition Specification, v1.4. Final release, 11/24/03, November 2003.

[Szy98] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1998.